

Crossprop: Learning representations through stochastic gradient descent in cross-validation error

Richard S. Sutton, Vivek Veeriah

Reinforcement Learning & Artificial Intelligence Lab
Dept. of Computing Science, University of Alberta
{ rsutton, vivekveeriah }@ualberta.ca

Introduction

Representations are fundamental to Artificial Intelligence. Typically, the performance of a learning system depends on its data representation. These data representations are usually hand-engineered based on some prior domain knowledge regarding the task. More recently, the trend is to learn these representations through deep neural networks as these can produce dramatical performance improvements over hand-engineered data representations. Learning representations reduces the human labour involved in any learning system design, and this allows in scaling of a learning system for difficult problems.

In this paper, we present a new incremental learning algorithm, called *crossprop*, for learning representations based on prior learning experiences. Unlike backpropagation, crossprop minimizes the cross-validation error. Specifically, our algorithm considers the influences of all the past weights on the current squared error, and uses this gradient for incrementally learning the weights in a neural network. This idea is similar to that of tuning the learning system through an offline cross-validation procedure.

Crossprop is applicable to incremental learning tasks, where a sequence of examples are encountered by the learning system and they need to be processed one by one and then discarded. The learning system can use each example only once and can spend only a limited amount of computation for an example.

The crossprop algorithm

The base learning system is a single hidden layer neural network, where n is the number of dimensions of the input vector and m is the number of hidden units (i.e. features). Since, we are concerned with a regression task in this paper, there is only a single output unit. However, our algorithm can be easily extended to classification tasks as well. A simple block diagram representing the structure of the base learning system is shown in Fig. 1.

The base learning system receives an example $X(t) \in \{0, 1\}^n$ at every time step. This input vector is mapped to the hidden units through the input weight matrix $U(t) \in \mathbb{R}^{n \times m}$ and a nonlinearity, like a logistic function or hyperbolic tangent, can be applied over this mapping. This results in the

Copyright belongs to the authors. All rights reserved.

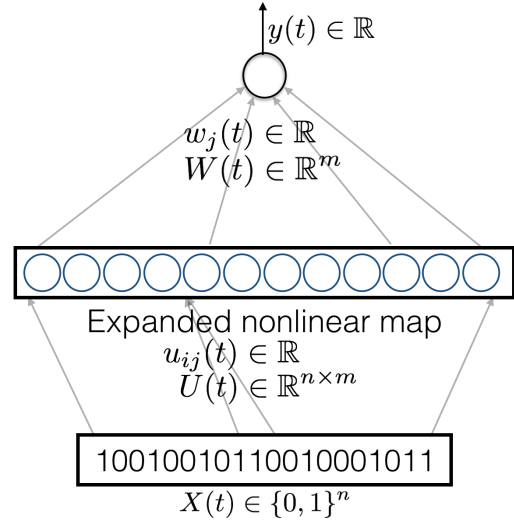


Figure 1: Base learning system: the input vector $X(t)$ is binary with a dimension of n . This is mapped onto m hidden units of a single layer neural network through the weights $U(t) \in \mathbb{R}^{n \times m}$ over which a nonlinearity, like the logistic or tanh function, is applied. Using these hidden units and the output weight vector $W(t) \in \mathbb{R}^m$, a real-valued output $y(t) \in \mathbb{R}$ is computed. The same learning system is referred throughout this paper. The hidden units of this base learning system are called *learnable features*. The number of dimensions of the input vector n is equal to 20. Here, there is only a single output unit as we are concerned with a regression task. However, extending our approach to a classification task is straightforward.

activations of the hidden units. For instance, when tanh function is used, then the activations can be mathematically expressed as, $\phi_j(t) = \tanh\left(\sum_{i=1}^n X_i(t)u_{ij}(t)\right)$, where $\phi_j(t)$ refers to the j th hidden unit in the base learning system, $X_i(t)$ refers to the i th element in the input vector and $u_{ij}(t)$ is the input weight connecting the i th input element to the j th hidden unit.

The hidden units $\phi_j(t)$ are successively mapped to form a single target output $y(t) \in \mathbb{R}$ of the base learning system

by the output weight vector $W(t) \in \mathbb{R}^m$. This can be defined as, $y(t) = \sum_{j=1}^m \phi_j(t)w_j(t)$, where $w_j(t)$ refers output weight connecting the j th hidden unit to the single output node.

The input and the output weights are incrementally learned through stochastic gradient descent. At each time step, a new example $X(t)$ is given to the base learning system, which is used to compute the output $y(t)$. In order to improve this estimate, the real target output $y^*(t)$ is also given to the base learning system from which a squared error $\delta(t)^2 = (y^*(t) - y(t))^2$ is computed, using the estimate and the target output. Subsequently, in crossprop, the gradients $\frac{\partial \delta(t)^2}{\partial w_j(t)}$ and $\frac{\partial \delta(t)^2}{\partial u_{ij}(t)}$ are computed. These are in turn used to improve the input and output mapping weights, by stochastic gradient descent.

Algorithm 1 Crossprop algorithm

INPUT: α, n, m

- 1: Initialize h_{ij} to 0
- 2: Initialize w_j and u_{ij} as desired
- 3: where $i = 1, 2, \dots, n; j = 1, 2, \dots, m$
- 4: **for** each new example $(X(t), y^*(t))$ **do**
- 5: $y \leftarrow \sum_{j=1}^m \phi_j(t)w_j(t)$
- 6: $\delta(t) \leftarrow (y^*(t) - y(t))$
- 7: **for** $j = 1, 2, \dots, m$ **do**
- 8: **for** $i = 1, 2, \dots, n$ **do**
- 9: $u_{ij}(t+1) \leftarrow u_{ij}(t) + \alpha \delta(t) \phi_j(t) h_{ij}(t)$
- 10: $h_{ij}(t+1) \leftarrow h_{ij}(t) \left(1 - \alpha \phi_j^2(t) \right) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}(t)}$
- 11: **end for**
- 12: $w_j(t+1) \leftarrow w_j(t) + \alpha \delta(t) \phi_j(t)$
- 13: **end for**
- 14: **end for**

Backpropagation is the most popular method for learning representations with neural networks. In backprop, $\frac{\partial \delta(t)^2}{\partial u_{ij}(t)}$ is computed and used for improving the input weights. By chain rule in calculus, this can be expressed as, $\frac{\partial \delta(t)^2}{\partial u_{ij}(t)} = \sum_{j=1}^m \frac{\partial \delta(t)^2}{\partial \phi_j(t)} \cdot \frac{\partial \phi_j(t)}{\partial u_{ij}(t)}$. The influence of each feature is considered while tuning the instantaneous input weights.

Unlike backprop, here we compute $\frac{\partial \delta(t)^2}{\partial u_{ij}(t)}$ instead of $\frac{\partial \delta(t)^2}{\partial \phi_j(t)}$. Specifically, we are concerned by the influences of all the past values of u_{ij} over the current instantaneous error $\delta(t)^2$. Learning or tuning the weights by considering all its past values is usually established by offline cross-validation methods. As our incremental learning algorithm achieves this effect without any additional offline computations, our method reduces the cross-validation error and so, our method is called *crossprop*. The complete algorithm is summarized in [1].

Derivation of crossprop

The objective of supervised learning is to learn parameters of a learning system in order to minimize the squared error $\delta^2(t)$, where $\delta(t) = y^*(t) - y(t)$, on future time steps.

Let n be the number of dimensions of an input sequence $X(t)$, such that $X(t) \in \mathbb{R}^n$ and let m be the number of units in the hidden layer.

$U(t)$ is defined as the weight matrix that maps the input sequence $X(t)$ to the features $\phi(t)$ (i.e., the hidden layer), which implies that $U(t) \in \mathbb{R}^{n \times m}$. Mathematically, this mapping is defined as $\phi_j(t) = \sigma\left(\sum_{i=1}^n X_i(t)u_{ij}(t)\right)$ where $u_{ij}(t)$ is an element belonging to the i th row and j th column of the matrix $U(t)$. σ is a logistic function and any other form of nonlinearity (e.g.: tanh, LTUs etc.) can be used here.

In turn, these features are mapped to an estimate $y(t) = \sum_{j=1}^m \phi_j(t)w_j(t)$ through a weight vector $W(t) \in \mathbb{R}^m$ where $w_j(t) \in W(t)$.

The least mean squares (LMS) learning rule updates the weights $w_j(t)$ at each time step as follows:

$$\begin{aligned} w_j(t+1) &= w_j(t) - \frac{1}{2} \alpha \frac{\partial \delta^2(t)}{\partial w_j(t)} \\ &= w_j(t) - \alpha \delta(t) \frac{\partial \delta(t)}{\partial w_j(t)} \\ &= w_j(t) - \alpha \delta(t) \frac{\partial [y^*(t) - y(t)]}{\partial w_j(t)} \\ &= w_j(t) + \alpha \delta(t) \frac{\partial y(t)}{\partial w_j(t)} \\ w_j(t+1) &= w_j(t) + \alpha \delta(t) \frac{\partial}{\partial w_j(t)} \left[\sum_{i=1}^m \phi_i(t) w_i(t) \right] \end{aligned}$$

$$w_j(t+1) = w_j(t) + \alpha \delta(t) \phi_j(t) \quad (1)$$

In order to learn the weights $U(t)$, we need to consider the influence of the past values of $U(t)$ on the current error $\delta^2(t)$. Specifically, we are concerned with finding $\frac{\partial \delta^2(t)}{\partial u_{ij}(t)}$ where u_{ij} includes all the past values of $u_{ij}(t)$.

This is interesting because most of the current research on representation learning usually consider only the influence of the weight at the current time step $u_{ij}(t)$ on the current squared error $\delta^2(t)$: $\frac{\partial \delta^2(t)}{\partial u_{ij}(t)}$. This ignores the effects of the previous possible values of these weights on the squared error at the current time step.

In order, to update the elements $u_{ij}(t) \in U(t)$, the update rule can be derived from the following equation:

$$\begin{aligned} u_{ij}(t+1) &= u_{ij}(t) - \frac{1}{2} \alpha \frac{\partial \delta^2(t)}{\partial u_{ij}(t)} \\ &= u_{ij}(t) - \alpha \delta(t) \frac{\partial [y^*(t) - y(t)]}{\partial u_{ij}(t)} \\ u_{ij}(t+1) &= u_{ij}(t) + \alpha \delta(t) \frac{\partial y(t)}{\partial u_{ij}(t)} \end{aligned} \quad (2)$$

Here, we consider the influence of all the past input weights $U(t)$ on the current estimate $y(t)$, through the output weights $W(t)$:

$$\begin{aligned}\frac{\partial y(t)}{\partial u_{ij}} &= \sum_k \frac{\partial y(t)}{\partial w_k(t)} \cdot \frac{\partial w_k(t)}{\partial u_{ij}} \\ \frac{\partial y(t)}{\partial u_{ij}} &\approx \frac{\partial y(t)}{\partial w_j(t)} \cdot \frac{\partial w_j(t)}{\partial u_{ij}}\end{aligned}\quad (3)$$

The approximation of $\sum_k \frac{\partial y(t)}{\partial w_k(t)} \cdot \frac{\partial w_k(t)}{\partial u_{ij}} \approx \frac{\partial y(t)}{\partial w_j(t)} \cdot \frac{\partial w_j(t)}{\partial u_{ij}}$ is reasonable because the primary effect on the input weight u_{ij} would be through the output weight $w_j(t)$.

By defining $h_{ij}(t) = \frac{\partial w_j(t)}{\partial u_{ij}}$, we can obtain a simple form for eqn. [3]:

$$\begin{aligned}\frac{\partial y(t)}{\partial u_{ij}} &\approx \frac{\partial y(t)}{\partial w_j(t)} \cdot \frac{\partial w_j(t)}{\partial u_{ij}} \\ &= \left(\frac{\partial}{\partial w_j(t)} \sum_k \phi_k(t) w_k(t) \right) \cdot h_{ij}(t) \\ \frac{\partial y(t)}{\partial u_{ij}} &\approx \phi_j(t) h_{ij}(t)\end{aligned}\quad (4)$$

$h_{ij}(t)$ is an additional memory parameter corresponding to the input weight $u_{ij}(t)$. Its update rule can be expressed as a recursive equation as follows:

$$\begin{aligned}h_{ij}(t+1) &= \frac{\partial w_j(t+1)}{\partial u_{ij}} = \frac{\partial}{\partial u_{ij}} \left[w_j(t) + \alpha \delta(t) \phi_j(t) \right] \\ h_{ij}(t+1) &= h_{ij}(t) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}} - \alpha \frac{\partial y(t)}{\partial u_{ij}} \phi_j(t) \quad (5) \\ &= h_{ij}(t) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}} - \alpha \left(\phi_j(t) h_{ij}(t) \right) \phi_j(t) \\ &= h_{ij}(t) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}} - \alpha \phi_j^2(t) h_{ij}(t) \\ h_{ij}(t+1) &= h_{ij}(t) \left(1 - \alpha \phi_j^2(t) \right) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}} \quad (6)\end{aligned}$$

By using eqn. [4] in eqn. [2], we can now define a recursive update equation for the weights $u_{ij}(t)$ and thereby summarize the complete algorithm:

$$\begin{aligned}u_{ij}(t+1) &= u_{ij}(t) + \alpha \delta(t) \phi_j(t) h_{ij}(t) \\ h_{ij}(t+1) &= h_{ij}(t) \left(1 - \alpha \phi_j^2(t) \right) + \alpha \delta(t) \frac{\partial \phi_j(t)}{\partial u_{ij}} \\ w_i(t+1) &= w_i(t) + \alpha \delta(t) \phi_i(t)\end{aligned}$$

Depending on the nonlinearity used for the hidden units, $\frac{\partial \phi_j(t)}{\partial u_{ij}}$ can be reduced to a closed-form equation.

For instance, if a logistic function is used, then $\phi_j =$

$$\sigma \left(\sum_{i=1}^n X_i(t) u_{ij}(t) \right),$$

$$\begin{aligned}\frac{\partial \phi_j(t)}{\partial u_{ij}} &= \frac{\partial \phi_j(t)}{\partial u_{ij}(t)} \\ &= \frac{\partial}{\partial u_{ij}(t)} \sigma \left(\sum_{i=1}^n u_{ij}(t) X_i(t) \right) \\ &= \sigma \left(\sum_{i=1}^n u_{ij}(t) X_i(t) \right) \left[1 - \sigma \left(\sum_{i=1}^n u_{ij}(t) X_i(t) \right) \right] X_i(t) \\ \frac{\partial \phi_j(t)}{\partial u_{ij}} &= \phi_j(t) \left(1 - \phi_j(t) \right) X_i(t)\end{aligned}$$

Another frequently used activation function is \tanh , which implies that $\phi_j = \tanh \left(\sum_{i=1}^n X_i(t) u_{ij}(t) \right)$,

$$\begin{aligned}\frac{\partial \phi_j(t)}{\partial u_{ij}} &= \frac{\partial \phi_j(t)}{\partial u_{ij}(t)} \\ &= \frac{\partial}{\partial u_{ij}(t)} \tanh \left(\sum_{i=1}^n u_{ij}(t) X_i(t) \right) \\ &= \left[1 - \tanh^2 \left(\sum_{i=1}^n u_{ij}(t) X_i(t) \right) \right] X_i(t) \\ \frac{\partial \phi_j(t)}{\partial u_{ij}} &= \left[1 - \phi_j(t)^2 \right] X_i(t)\end{aligned}$$

Online supervised learning

We consider an online supervised learning setting for our experiments where data arrives as a sequence of examples. The k th example is presented as a vector of n binary inputs $X(t) \in \{0, 1\}^n$ with each element $X_i(t) \in \{0, 1\}$, $i = 1, 2, \dots, n$ and a single target output $y^*(t) \in \mathbb{R}$. Here the learning system needs to learn a function that maps the input to the target output in an online manner, i.e., the learner can use each example only once and can spend a limited amount of computation for each incoming example.

GENERIC Online Feature Finding (GEOFF) task

In order to evaluate crossprop and various other representation learning methods, we introduce the generic online feature finding (GEOFF) task.

The data in our experiment was generated through simulation as a series of examples of 20-dimensional i.i.d. input vectors (i.e., $n = 20$) and a scalar output target. Inputs $X(t)$ were binary, chosen randomly between 0 and 1 with equal probability, i.e., $X(t) \in \{0, 1\}^n$. The target output was computed by linearly combining 50 target features $\phi^*(t)$, which were generated from the inputs using 50 fixed random Linear Threshold Units (LTUs) (i.e., $m = 50$). These features $\phi^*(t)$, which are used to generate the target output $y^*(t)$, are called the *target features*.

This nonlinear map from the input to the features is achieved using Linear Threshold Units (LTUs). This particular form of representation is adopted from Sutton and

Whitehead’s (1993) work, where each feature is computed as follows:

$$\phi_j^*(t) = \begin{cases} 1, & \sum_{i=1}^n X_i(t)u_{ij}^* > \theta_j \\ 0, & \text{otherwise} \end{cases}$$

where $X_i(t)$ is the i th element of the input vector $X(t)$ and u_{ij}^* is the input weight for the i th input element and the j th feature (i.e., hidden unit). $\phi_j^*(t)$ refers to the j th target feature.

The input weight matrix $U^* \in \{-1, 1\}^{n \times m}$ consists of elements $u_{ij}^* \in \{-1, 1\}$, which is chosen randomly. u_{ij}^* maps the i th input element with the j th feature and remains fixed after initialization. The threshold θ_j is initialized in such a way that the j th feature is activated only when at least β proportion of the input bits matches the prototype of the feature. This can be achieved by setting the threshold as $\theta_j = n\beta - S_j$, where S_j is the number of input weights with a value of -1 , connected to the j th feature. The threshold parameter β for the LTUs was set to 0.6.

The target output $y^*(t)$ was then computed as a linear map from the target features $\phi_j^*(t)$ as $y^*(t) = \sum_{j=1}^m \phi_j^*(t)w_j^* + \epsilon(t)$, where $\epsilon(t) \sim N(0, 1)$ is a random noise. The target output weights w_j^* were randomly chosen between $-1, 0$ and 1 with equal probability. Their values were chosen once and kept fixed for all examples. As these values remain fixed this task is also called as *stationary* GEOFF task. The learner can observe only the inputs $X(t)$ and output $y^*(t)$.

If the features $\phi_j(t)$ and the output weights w_j of a learner are equal to that of the target network respectively, then the MSE performance $\mathbb{E}[(y^*(t) - y(t))^2]$ of the learner would be minimum, which is 1. Therefore, the task of representation learning is to learn these weights u_{ij}^* and w_j^* , in order to produce similar features.

Experiments

In this section, we study the performances of crossprop and backprop on the stationary GEOFF task.

The backpropagation algorithm is the standard and the most popular method for learning representations. We use online backpropagation algorithm to minimize the squared error $\delta(t)^2 = (y^*(t) - y(t))^2$, where online backpropagation uses a stochastic gradient descent rule to learn both input and output mapping weights.

Experiment 1: Multiple target and learnable feature settings

We experimented with the number of target and learnable features for both the algorithms. For each setting of the number of target and learnable features, we swept over a set of step-sizes: $\{10^{-4}, 5 \times 10^{-4}, \dots, 0.1\}$ and chose the step-size that gave the best asymptotic MSE performance for the backpropagation algorithm. This was then used to generate the learning plots for crossprop and other variants of backprop. Hyperbolic tangent function was used as the activation function for all the learning methods. Each experiment was repeated for 50 independent runs and the averaged results are plotted here.

Crossprop, backprop and backprop with momentum were empirically studied on the stationary GEOFF task. In this experiment, the number of target features (i.e., number of LTUs in GEOFF task) and the number of learnable features were varied and the learning rates of these methods were observed. We considered three settings: the first one had 50 target features and 100 learnable features; the second setting had 500 target features and 1000 learnable features; the last setting had 1000 target features and 2000 learnable features. The learning curves with mean squared errors and squared errors for these settings are shown in Fig. [2].

When task had only 50 target features (Fig. 2 (a, d)), backprop with momentum learned the features much faster than crossprop and backprop. This was a simple setting and there was not much room for crossprop to outperform the conventional representation learning methods. When the number of target features were increased to 500 (Fig. 2 (b, e)), the variant of backprop clearly diverged, while both crossprop and backprop had similar learning curves. This trend continued when the number of target features increased further to 1000, where crossprop had a clear advantage in terms of the learning speed when compared to the other two methods.

Experiment 2: Different step-sizes and their effects

In the previous section, we used the same step-size parameter for updating both input weights u_{ij} and the output weights w_j . However, it is interesting to study the effects of different step-sizes for these weights. For instance, in certain nonstationary learning tasks, the features need to change *slowly* over time while the estimates have to adapt quickly to the changes. Having different step-sizes for the weights is a simple way of achieving this.

In the following experiments, the step-size in eqn. [1] is termed α whereas the step-size in eqns. [2 & 6] are called β . The same range of step-sizes are considered as in the previous experiment. The learning curves obtained from this experiment is shown in Fig. [3].

In Fig. [3 (a)], the step-size for output weights is set to 0.0001 whereas for Fig. [3 (b)] the step-size is set to 0.001. The target features are 500 features for both these plots. It can be observed that, when α is larger, crossprop method seems to perform better than backprop and its variant. This is an interesting result because, in applications where nonstationarity needs to be tracked, it is important to have a larger step-size without causing the learning to diverge. Crossprop seems to be relatively stable than backprop for larger α unlike backprop.

Conclusion

In this paper, we presented a *different* approach towards learning representations and this approach is called crossprop. From our preliminary results, crossprop was observed to have a better learning rate compared to backprop and its variants on difficult versions of the GEOFF task. Moreover, it had relatively stable learning with a faster step-size compared to backprop. This implies that crossprop is better suited in nonstationary tasks unlike backprop. Our future work includes experiments in *nonstationary* GEOFF tasks and MNIST dataset.

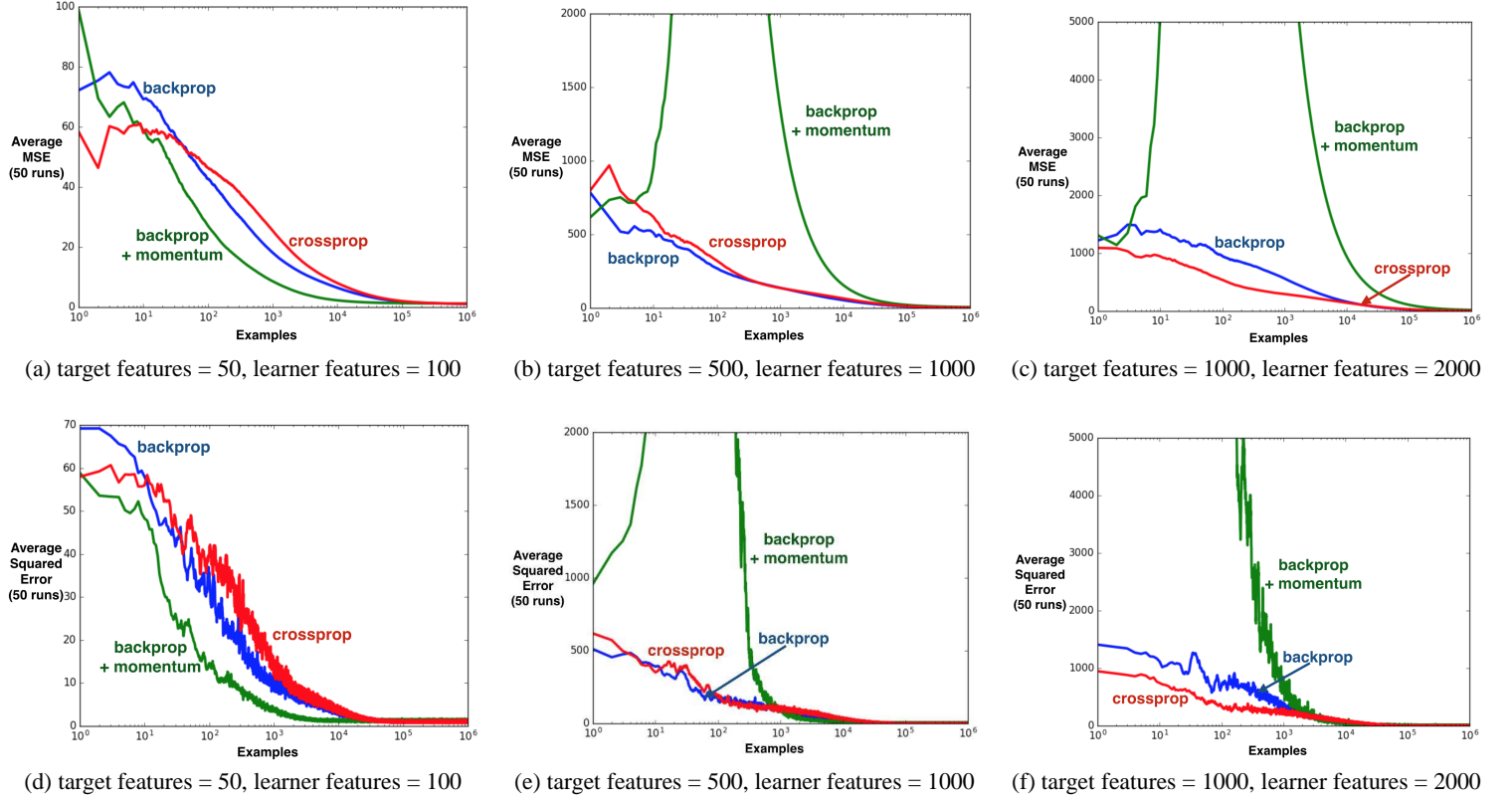


Figure 2: (a, b, c) are plots of mean squared error over the number of examples experienced by each learning algorithm. Similarly, (d, e, f) are plots of squared error. When the number of target features is 50, the representations are learned quickly by the backprop methods. On increasing this number of target features to 500, it can be observed that the backprop and crossprop methods have relatively similar performance in terms of both mean squared and squared errors. The real advantage of crossprop can be observed when the number of target features increases further (i.e., 1000 target features). In this difficult setting, crossprop significantly outperforms both backprop and backprop with momentum, which is the key result. Adding momentum to backprop improves the performance in smaller settings of the GEOFF task (i.e., target features = 50). However, this dramatically deteriorates the performance when the task becomes increasingly difficult.

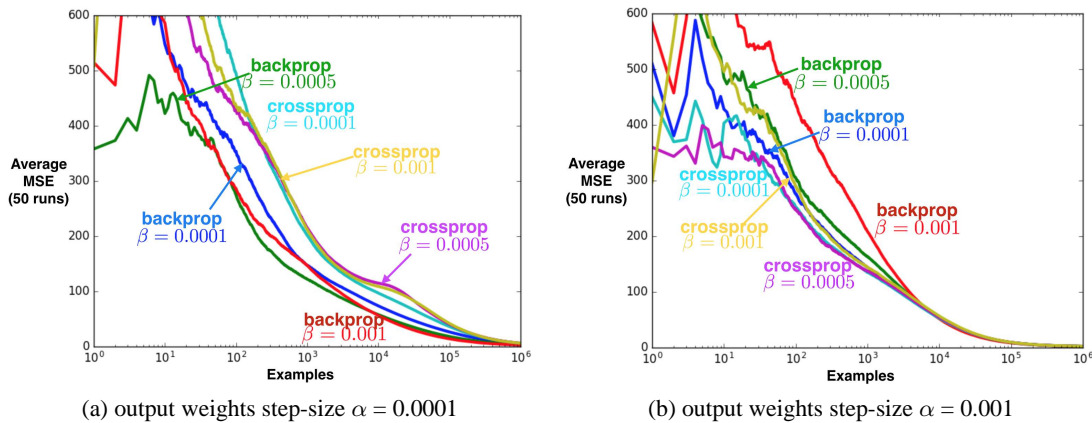


Figure 3: α is the step-size that controls the update of the output weight vector $W(t)$ in the learning system. β is the step-size that controls the update of the input weight matrix $U(t)$. When there are 500 target features and 1000 learnable features, it can be observed that a relatively higher value of α seems to improve the learning rate of crossprop while this produces a poor performance in the backprop method.